DSA Tutorial Week 5 - Graphs

Find All Source Code on the DSA Tutorial Github: DSA-Tutorial-Exercises

---

Problem 0 - Preparation !examine before tutorial!

You are given the following description of an undirected graph:

```
Vertices:   A, B, C, D, E
Edges:      AB, AC, BD, CD, DE
```

a. Represent this graph using an *adjacency matrix*, an *adjacency list*, and an *edge list*. Illustrate them using pen and paper.
b. After drawing/writing the graph in all three forms, analyze the space complexity of each representation and reflect on which type of graph (sparse or dense) is best suited for each representation.
c. Consider how different operations such as adding or removing edges, and finding adjacent vertices, are affected by the choice of representation.
d. To deepen your understanding and to verify your solution, use visualgo (https://visualgo.net/en/graphds) to visualize the graph in each representation form and compare it with your implementation.

Problem 1 - Finding Friends

a. You are given a set of persons **P** and their friendship relation **R**. That is, **(a, b)** ∊ **R** if and only if **a** is a friend of **b**. You must find a way to introduce person **x** to person **y** through a chain of friends.

   Model this problem with a graph and describe an algorithm to solve the problem.

   Pointers

   - First, think of a method that you know which solves the underlying, more general problem. If it is a shortest-path problem, should we traverse the vertices in a certain way?
   - To find the more general problem, think of what we are trying to achieve with the graph and try to express it in terms of collections of vertices and edges.
   - Now think of which adjustments are needed to the method to solve this specific instance of the more general problem.

b. If you were to implement this algorithm, which graph implementation would you choose for this problem? Why?

   Pointers

   - Determine how this problem can be modeled as a search for a path in a graph.
   - Consider using a graph traversal algorithm like Breadth-First Search (BFS) or Depth-First Search (DFS). Which would be most appropriate for this problem? What will that mean for your choice?

c. Implement the algorithm in java. Use ***FriendsChainFinder.java*** as a starting point. A solution is provided in the “answers” directory.

Reflection

- Reflect on how modeling real-world problems as graphs can provide clarity and a structured approach to problem-solving.
- Consider the modifications made to the standard algorithm to track the path. How did these changes enable the algorithm to solve the specific problem at hand?
- Reflect on the choice of an adjacency list for graph representation. How does this choice impact the algorithm's performance, especially in terms of space and time complexity?

Problem 2 - Difficult Parties

Consider a graph **G = (V, E)** which represents a social network in which each vertex represents a person, and an edge **(u, v)** ∈ **E** represents the fact that **u** and **v** know each other.

Your problem is to organize the largest party in which **nobody knows each other**. This is also called the *maximal independent set* problem.

Formally, given a graph **G = (V, E)**, find a set of vertices S of maximal size in which no two vertices are adjacent. (I.e., for all **u** ∈ **S** and **v** ∈ **S, (u, v)** ∉ **E**)

a. Define an algorithm for the Maximal Independent Set problem above. Write the algorithm in pseudocode.

   Pointers:

   - Begin by clarifying the definition of an independent set in a graph: a set of vertices where no two vertices are connected by an edge. The goal is to find the largest such set, known as the maximal independent set.
   - Consider an approach to identify independent sets. *For instance*, iterate through vertices and decide whether to include each vertex in the independent set based on its connections.
   - Implement a method to check if adding a vertex to the current set maintains the independence of the set.
   - Develop a strategy for incrementally constructing the independent set. This could involve greedy methods, backtracking, or other approaches, depending on the complexity you want to introduce.

b. Write a verification algorithm in pseudocode for the maximal independent set problem. This algorithm, called `testIndependentSet(G, S)`, takes a graph **G** represented through its adjacency matrix, and a set **S** of vertices, and returns true if **S** is a valid independent set for **G**.

c. Reflect on the time complexity of your Maximal Independent Set algorithm. What factors contribute to its complexity? Consider the process of iterating through vertices, checking for edges, and constructing the independent set. What do you think happens when you run your algorithm on a large graph?

d. Analyze the time complexity of the `testIndependentSet` function. What are the key operations in this function, and how do they contribute to its overall complexity? Does the size of the graph have a large impact on this algorithm? Does this result surprise you, when compared to the reflection in c.?

Problem 3 - Universal Sink

In a directed graph **G**, a **universal sink** is a unique vertex that has incoming edges from all other vertices and no outgoing edges. This means every other vertex in the graph points to the universal sink, but the universal sink points to none. The concept of a universal sink is especially interesting because if one exists in a graph, *it is the only one*.

Example
Consider the graph below:

$$\begin{tikzcd}
0 \arrow[r] \arrow[d] & 1 \arrow[dl] \\
2 & 3 \arrow[l]
\end{tikzcd}$$

- Vertex 0 has outgoing edges to vertices 1 and 2 but no incoming edges.
- Vertex 1 has an incoming edge from 0 and an outgoing edge to 2.
- Vertex 2 has an incoming edge from 0, 1 and 3 (no outgoing edges).
- Vertex 3 has an outgoing edge to 2 but no incoming edges.

Vertex 2 is a universal sink because it has incoming edges from all other vertices (0, 1 and 3) and no outgoing edges.

Design and implement an algorithm that tests if a graph ***G***, represented as an **adjacency matrix**, has a universal sink. First write your algorithm in pseudocode, then write it in Java. Your algorithm should run in ***O(n)*** time. For this problem, use an adjacency matrix implementation.

Reflection
- The key insight is that if a universal sink exists, there can be only one such vertex. What does this mean for the algorithm?
- How can you find out if the node is the "source of no edges"?
- After writing the algorithm, think about the significance of using two pointers (i and j) to traverse the adjacency matrix. Why does incrementing i when `G[i][j]` is `1` (and `j` when it's `0`) effectively lead you to a potential universal sink?
- How can you show that your algorithm runs in ***O(n)*** time? Can you think of a worst-case input?